Implementação e avaliação da cifra de fluxo Forro14 em hardware programável Tofino usando a linguagem P4

Rodrigo A. de A. Pierini[1], **Caio Teixeira**[1],
Christian Esteve Rothenberg[1], **Marco A. Amaral Henriques**[1]

[1] Faculdade de Engenharia Elétrica e de Computação
Universidade Estadual de Campinas
Campinas - SP - Brasil

{rpierini,chesteve,maah}@unicamp.br

caio@dca.fee.unicamp.br

Resumo. *O paradigma de redes definidas por software (SDN) habilitou diversas inovações em redes de computadores, principalmente na programabilidade do processamento de pacotes. Neste trabalho, investigou-se a viabilidade e os impactos em recursos computacionais do algoritmo de cifra de fluxo Forro14 em hardware de switch programável Tofino usando a linguagem P4. Para fins de comparação, foi analisado também o algoritmo ChaCha20 quanto a seu desempenho e impacto no mesmo switch. Constatou-se que o algoritmo Forro14 tem um desempenho melhor usando menos recursos que o ChaCha20 para comunicações de até 10 Gbps. Entretanto, quando são adotadas técnicas de paralelização, ChaCha20 tem um desempenho melhor para taxas maiores de dados, mas utilizando mais recursos de processamento do dispositivo que Forro14.*

Abstract. *The software-defined networking (SDN) paradigm has enabled several innovations in computer networking, especially in programmable packet processing. This paper investigated the feasibility and impact on computing resources of the Forro14 stream cipher algorithm in the Tofino programmable hardware switch. For comparison purposes, the ChaCha20 algorithm was also analyzed in terms of its performance and impact on the same switch. It was observed that the Forro14 algorithm performs better using fewer resources than ChaCha20 for communications up to 10 Gbps. However, when parallelization techniques are adopted, ChaCha20 performs better for higher data rates but uses more processing resources than Forro14.*

1. Introdução

Nas últimas décadas, as redes de computadores passaram por mudanças de paradigma relevantes para cenários onde a gestão da rede e sua operação possuem maior importância financeira e maior desempenho, como em redes metropolitanas e redes de *data centers*. Essas mudanças caracterizam-se pela separação entre o encaminhamento de pacotes (plano de dados) e a inteligência que define a forma de fazer esse encaminhamento (plano de controle). Esse é o paradigma de redes de computadores definidas por software (*Software-Defined Networking*, SDN) [Kreutz et al. 2014] que traz a possibilidade de centralizar o controle de uma rede e utilizar software de controle que definem

formas inovadoras de encaminhamento, possíveis pela programabilidade do plano de dados usando por exemplo o protocolo OpenFlow [Fernandes and Rothenberg 2014], ou mais recentemente a linguagem P4.

As tecnologias habilitadoras de SDN trazem diversas novas possibilidades de inovação não só em engenharia de tráfego, processamento de pacotes e otimização de recursos computacionais, mas também em segurança de redes de computadores. Nessa linha, novas formas de trazer garantias de segurança para as redes vêm sendo exploradas em diversas frentes, como detecção de intrusão [Mahrach et al. 2018], *firewall* [Datta et al. 2018], detecção de ataque de negação de serviço [Sivaraman et al. 2017], IP *Spoofing* [Li et al. 2019] e até mesmo operações de *hash* [Yoo and Chen 2021] e criptografia [Chen 2020] diretamente no encaminhamento de pacotes. Tais implementações possuem impactos no desempenho e nos recursos computacionais dos equipamentos de comutação. Portanto, busca-se soluções que provejam as garantias de segurança com o menor impacto nos recursos e desempenho.

Dentro desse contexto, este trabalho tem por objetivo a implementação e avaliação de uma cifra de fluxo brasileira chamada Forro14 [Coutinho et al. 2023b] em SDN, de forma a se verificar sua viabilidade em comparação com outra solução de sigilo nos critérios de vazão de dados e ocupação de recursos. Até onde sabemos, esta é a primeira iniciativa de implementar e avaliar tal cifra em um *switch* programável baseado em *hardware* Tofino e usando a linguagem P4 [Bosshart et al. 2014].

Aplicar esses algoritmos diretamente em equipamentos de encaminhamento de pacotes pode reduzir o uso de recursos de processamento de servidores, permitir a distribuição de processamento através da rede e até aproveitar a posição física de equipamentos para a setorização de processamento de dados. Nesse sentido, este trabalho é parte de um esforço maior para o desenvolvimento de um esquema de atestação remota distribuída e granular que permite a verificação de integridade em um contexto de *data centers* sem a oneração de um verificador central, de forma que os próprios *switches* da rede sejam capazes de verificar a integridade dos dispositivos conectados a ele. Isso exige que o *switch* seja capaz de calcular *hashes* ou cifras com o menor impacto possível em suas demais operações de encaminhamento e, consequentemente, faz-se necessário usar a linguagem P4.

É necessária a escolha de um algoritmo de cifra nesse contexto, o que nos leva à seguinte pergunta de pesquisa para este trabalho:*"No contexto de redes SDN baseadas em linguagem P4, a cifra de fluxo Forro14 é uma alternativa mais eficiente que a cifra ChaCha20 em uso de recursos (memória, portas etc.) e em vazão de dados?"*.

Para avaliação do algoritmo de cifra de fluxo destacado, foi feita uma implementação utilizando a tecnologia de plano de dados programável em um *switch* Tofino, e uma avaliação de desempenho com geração controlada de tráfego e coleta de métricas de vazão e utilização dos recursos do *switch*. A avaliação foi complementada pela comparação com a implementação de outra cifra de fluxo (ChaCha20), bastante conhecida na literatura e base para o projeto do Forro14.

2. Programabilidade do Plano de Dados com a linguagem P4

Um plano de dados programável é uma tecnologia que pode ser compreendida como uma especialização de SDN. Nos primeiros trabalhos com SDN, a abordagem utili-

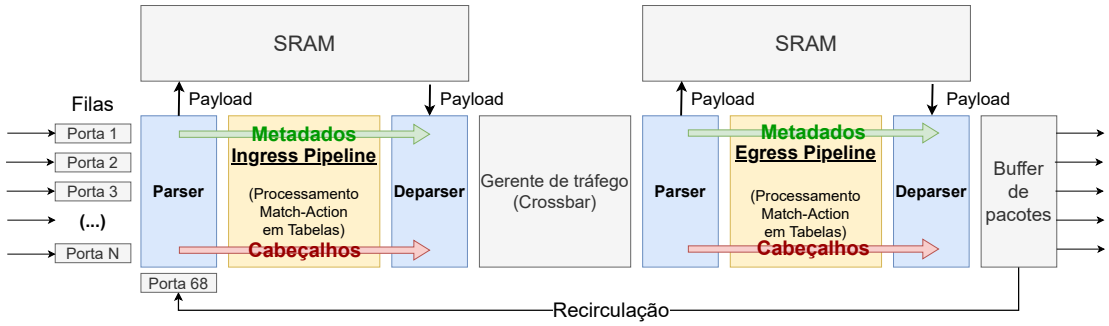

Figura 1. Arquitetura referência de um *switch* programável com P4. Fonte: Adaptado de *Systems Approach*. [Peterson et al. 2021].

zada se baseava no protocolo OpenFlow [Fernandes and Rothenberg 2014], um protocolo em que fabricantes de *switches* estruturam o *chip* de encaminhamento dos equipamentos de forma que algumas funcionalidades pré-estabelecidas no protocolo possam ser programadas no dispositivo por um controlador central.

Embora isso permita que diversas funcionalidades do *switch* possam ser controladas de forma centralizada, isso ainda limita o processamento de pacotes às definições do *chip* de encaminhamento do *switch* que é construído através de protocolos pré-conhecidos no contexto de rede, como *Ethernet*, IP, entre outros.

Para acelerar a inovação e liberar o operador das limitações de fabricantes de *switches*, surgiu o conceito de plano de dados programável. Nessa tecnologia, o fabricante define um *chip* de encaminhamento que pode ser programado, de forma similar à programação de uma FPGA (*Field Programmable Grid Array*) e trabalha na velocidade de linha do equipamento. O usuário é responsável por definir os protocolos e as operações que o *switch* deverá executar antes da sua operação na rede. O modelo de programação adotado para essa abordagem é o uso de tabelas *Match-Action*, onde o *switch* é compreendido como uma máquina de busca em tabelas e manipulação de dados em estruturas pré-definidas, direcionando os mesmos de acordo com os resultados das buscas.

Para realizar essa programação do chip, o usuário utiliza uma linguagem de domínio específico conhecida como P4 (*Programming Protocol-independent Packet Processors*) [Bosshart et al. 2014]. O código em alto nível definido pelo usuário é acrescido de uma definição de arquitetura fornecida pelo fabricante do *switch* e inserido em um compilador que gera o código compatível com a programação do *chip*, na forma de bytecode ou JSON (*JavaScript Object Notation*). A Figura 1 mostra uma arquitetura de referência do tipo *protocol-independent switch architecture* (PISA) que pode ser programada em P4.

A linguagem P4 se tornou um padrão para a programação de processadores de pacotes e é adotada em diversos tipos de plano de dados, como *switches* programáveis, SmartNICs [Scholz et al. 2019] e até plano de dados *in-kernel* como eBPF [Vieira et al. 2020].

Com o uso desses planos de dados programáveis, diversas aplicações mais complexas puderam ser implementadas dentro da própria infraestrutura de encaminhamento de dados, o que viabilizou um novo paradigma chamado *in-network computing* [Tokusashi et al. 2018], onde diversas operações de computação po-

dem ser executadas diretamente no plano de dados, como algoritmos de consenso de sistemas distribuídos [Dang et al. 2020], *cache* em rede [Jin et al. 2017], aprendizado de máquina [Zheng et al. 2023], entre muitas outras [Kfoury et al. 2021, Hauser et al. 2023].

No entanto, a linguagem P4 tem uma série de limitações que tornam desafiadora a implementação de algoritmos não diretamente ligados a tráfego de pacotes. Um exemplo sempre é dado: P4 não oferece suporte a laços de repetição. Uma nova iteração sobre dados em processamento é obtida com a reintrodução (recirculação) do pacote que contém tais dados. Por isso, implementações de cifradores como o AES e o ChaCha20 são complexas e exigem técnicas que são dependentes do hardware onde o processo será executado.

3. Cifras de Fluxo

Com o objetivo de prover sigilo em uma comunicação, algoritmos de criptografia podem ser utilizados para garantir que apenas detentores de uma chave criptográfica possam conhecer o conteúdo original de uma mensagem cifrada, sendo computacionalmente inviável a decifração do texto cifrado sem o conhecimento da chave.

Dentre os algoritmos de criptografia simétrica existentes, dois grandes grupos são definidos: *algoritmos de cifra em blocos*, nos quais blocos de dados de tamanho fixo são cifrados, e *algoritmos de cifra de fluxo*, nos quais os bits são cifrados um a um usando um fluxo gerado a partir de uma chave.

No contexto de algoritmos de cifra em blocos utilizados em redes programáveis, destaca-se a implementação do relevante algoritmo AES [Dworkin et al. 2001], com desempenho satisfatório em diversas aplicações. O algoritmo foi implementado utilizando uma técnica de *scrambled lookup tables*, onde os conteúdos das *S-Boxes* são previamente calculados de forma a simplificar partes do processo de cifração e decifração, usando buscas em tabela [Chen 2020].

Já no contexto de cifras de fluxo, um algoritmo comumente utilizado é o ChaCha20 [Bernstein et al. 2008]. Esse algoritmo utiliza uma chave e um *nonce* para gerar uma matriz de estado 4x4 com índices de 32 bits, totalizando 512 bits, que é utilizada na cifração do conteúdo da mensagem em claro com uma operação de XOR bit a bit. Para que o algoritmo seja seguro, é necessário que cada matriz de estado calculada seja única, não permitindo que uma mesma matriz de estado seja utilizada para cifrar dois conjuntos de 512 bits da mensagem. Para isso, são utilizados *nonces* e contadores, com o *nonce* variando a cada mensagem e o contador variando a cada conjunto de 512 bits de uma mesma mensagem.

Esse algoritmo utiliza 20 rodadas de operações divididas em funções chamadas QR (*Quarter Round*), onde 4 QRs compõem uma rodada do algoritmo. Os QRs são operações de somas módulo 2^{32} e XORs seguidos de rotações de bits (conjunto conhecido por ARX, *Add-Rotate-Xor*) que causam a difusão das modificações nos bits por toda a matriz de estado. Os QRs são aplicados em colunas da matriz de estado em rodadas ímpares e nas diagonais em rodadas pares. A Figura 2 demonstra as operações de cada QR e como os elementos de 32 bits da matriz de estado são coletados para cada QR. Os elementos C_i, K_i, T_i e N_i da matriz inicial correspondem, respectivamente, a **constante,**

chave, contador e **nonce**. A matriz final após 20 rodadas é somada (módulo 2^{32}) elemento a elemento com a matriz inicial para produzir os bits do fluxo de chave (*keystream*) que será usado para fazer o XOR com um conjunto de 512 bits do texto em claro (mensagem). Para os próximos conjuntos de 512 bits da mensagem, uma nova matriz de estado deve ser calculada com um incremento em um contador, gerando resultados diferentes a cada nova iteração.

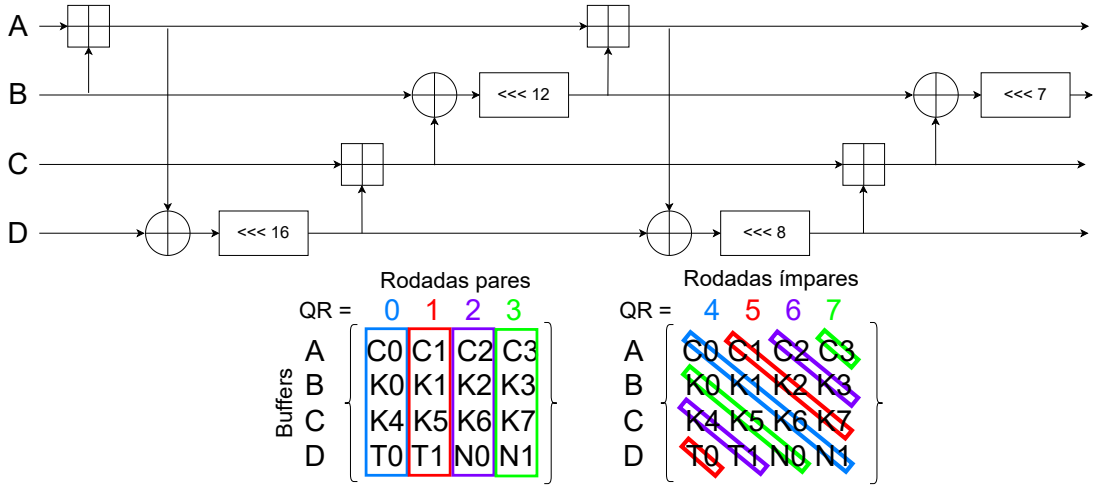

Figura 2. Esquema do Quarter Round e do uso dos elementos de 32 bits da matriz de estados no ChaCha20.

No contexto de SDN, o algoritmo ChaCha20 foi implementado em *switches* programáveis de forma paralela, sendo capaz de cifrar conjuntos de 512 a 3072 bits com vazão de 50 a 100% maior que a da implementação do algoritmo de cifra de blocos AES com chave de 256 bits [Yoshinaka et al. 2022].

Outro algoritmo de cifra de fluxo proposto recentemente é o chamado Forro14 [Coutinho et al. 2023b]. Esse algoritmo utiliza uma estrutura muito similar à do ChaCha20 e adiciona uma técnica chamada *polinização* para aumentar a difusão da matriz de estado calculada, de forma a torná-la mais segura contra ataques de análise diferencial. Essa técnica torna a implementação do algoritmo sequencial, visto que a polinização cria uma dependência ao QR atual utilizar um elemento alterado no QR anterior, limitando a velocidade de operação do algoritmo por não permitir o aproveitamento de recursos de paralelização de mais baixo nível que possam estar disponíveis em certos dispositivos. A Figura 3 mostra as operações de cada QR do Forro14 e como os valores da matriz de estado são coletados a cada QR. Observa-se uma reorganização dos elementos C_i, K_i, T_i e N_i da matriz inicial em comparação ao ChaCha20. O *buffer* E (pólen) é coletado de forma circular na primeira linha da matriz, começando pelo campo K_3 na última coluna.

Devido à técnica de polinização (parâmetro E), o algoritmo Forro14 requer menos processamento para ter uma segurança equivalente à do algoritmo ChaCha20, com uma redução de aproximadamente 30% no número de rodadas. No Forro14, são utilizadas mais operações de soma e menos operações de rotação de bits e XOR, tendo sido as rotações escolhidas de forma experimental para maximizar a segurança contra ataques de análise diferencial.

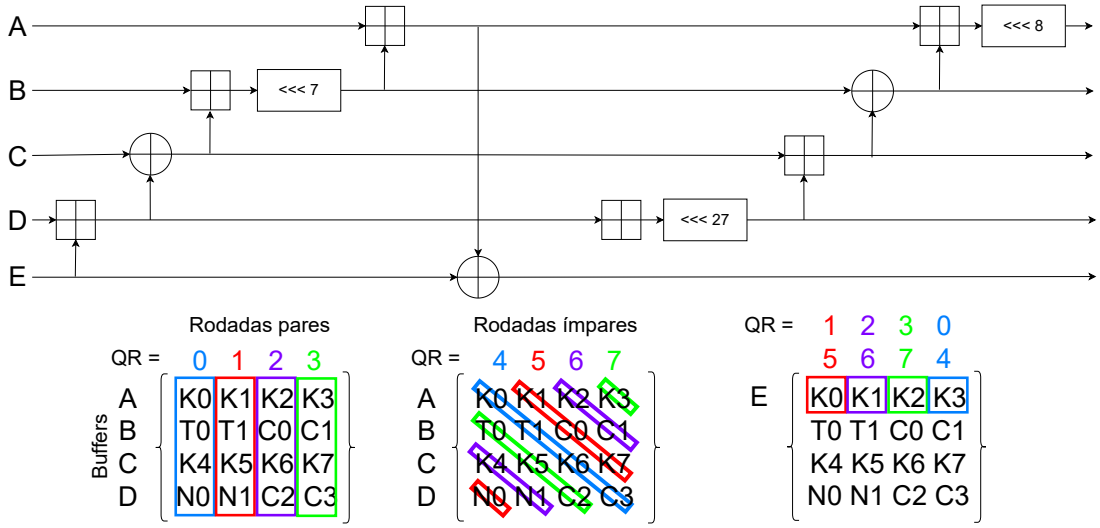

Figura 3. Esquema do *Quarter Round* e do uso dos elementos de 32 bits da matriz de estados do Forro14.

4. Implementação e avaliação do Forro14

Neste trabalho foi feita a implementação do algoritmo Forro14[1] e a avaliação de seu desempenho em *switches* programáveis em comparação com implementações paralela e sequencial do algoritmo ChaCha20.

A plataforma de SDN escolhida foi a ASIC (*Application-Specific Integrated Circuit*) Tofino da Intel. Esse ASIC programável permite o uso de *pipelines* de entrada e saída de 12 estágios, totalizando 24 estágios de processamento. O *switch* que embarca esse ASIC é um Edgecore Wedge 100BF-32X modelo DCS800 com 32 portas de 100 Gbps QSFP28.

A plataforma Tofino possui diversas limitações em sua configuração para que os estágios de pipeline trabalhem na velocidade de 100 Gbps sem gargalos. Como premissa da arquitetura de *switches*, não é possível utilizar estruturas de *loop* no processamento de pacotes, sendo necessária a recirculação de um mesmo pacote para esse fim. Com isso, o *switch* disponibiliza 2 portas internas utilizadas para recirculação de pacotes. Também é possível definir portas físicas como portas de *loopback* para recirculação.

A estrutura do pipeline do ChaCha20 implementado de forma paralela permite aproveitar uma travessia de um dos dois *pipelines* (*Ingress* ou *Egress*) para realizar 4 QRs juntos, visto que não há interdependência entre os resultados dos QRs. Portanto, com uma travessia no *pipeline* de entrada (*Ingress*) e uma travessia no *pipeline* de saída (*Egress*), é possível realizar 2 rodadas do algoritmo. Com 10 recirculações do pacote através de portas internas do próprio *switch*, é possível realizar as 20 rodadas do ChaCha20. Para controle do estado entre as recirculações, a implementação espera receber pacotes com um novo cabeçalho após o Ethernet, contendo 3 informações: modo de operação (cifração ou decifração), número da rodada e um índice (de 0 a 5) do conjunto de 512 bits que está sendo processado do *payload* (mensagem), conforme a Figura 4.

[1]https://github.com/regras/p4-forro

Quando em modo de cifração, a implementação do ChaCha20 paralelo realiza a geração de um *nonce* no próprio Tofino, enquanto no modo decifração é utilizado um *nonce* informado junto à mensagem cifrada. Para simplificação dos testes de desempenho, todos os algoritmos usaram um *nonce* fixo para realizar a decifração de uma mensagem gerada aleatoriamente.

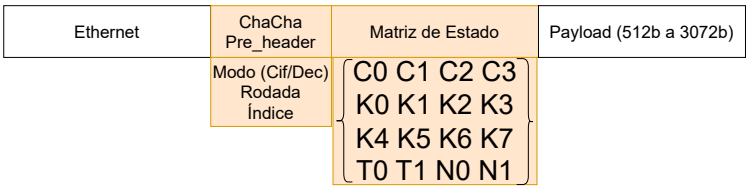

Figura 4. Estrutura de cabeçalhos do pacote de processamento do ChaCha20 paralelo em Plano de Dados Programável.

Em uma implementação direta, o algoritmo realiza as 12 operações básicas ao processar cada QR e assim o *pipeline* fica totalmente ocupado para o processamento, não sendo possível a inicialização e a finalização da matriz de estado, nem a cifração/decifração do conteúdo (o XOR do conjunto de 512 bits com a matriz). Entretanto, utilizando uma instrução especial da arquitetura Tofino chamada *Identity Hash*, é possível realizar uma operação de rotação de bits e uma operação de soma módulo 2^{32} ou XOR em um mesmo estágio com algumas adaptações no código P4. Com isso, é possível fazer a inicialização junto da primeira rodada e a finalização junto da última rodada, sendo necessária apenas uma recirculação a mais para se obter o cálculo do XOR com a matriz de estado. A Figura 5 mostra as travessias do pacote para o cálculo da matriz de estado.

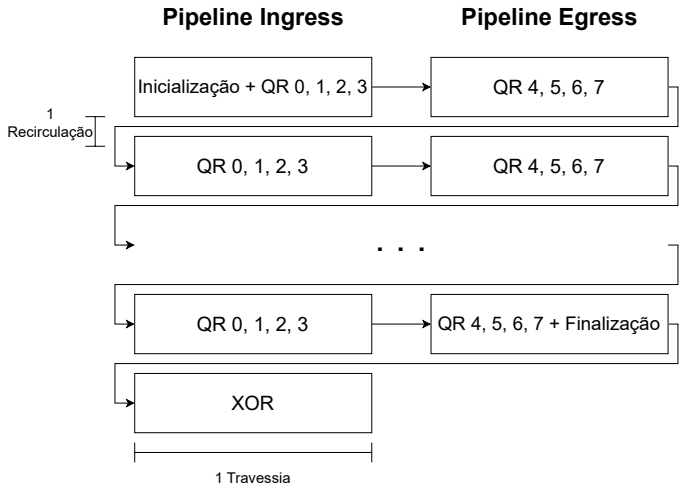

Figura 5. Algoritmo ChaCha20 paralelo em modo decifração em Plano de Dados Programável.

Essa mesma técnica foi utilizada para a implementação do algoritmo Forro14. No entanto, devido à natureza sequencial do algoritmo, não foi possível fazer o processamento de mais de um QR no mesmo *pipeline*, o que implica em quatro travessias (com uma recirculação entre cada duas travessias) para calcular os quatro QRs que equivalem

a uma rodada do algoritmo. Com isso, são necessárias 28 recirculações para as 14 rodadas, o que resulta em 56 travessias para o cálculo da matriz de estado e uma extra para a cifração/decifração da mensagem. A quantidade de recirculações afeta diretamente a vazão que o algoritmo pode alcançar, visto que o mesmo pacote precisa passar várias vezes pelo *pipeline* e que as portas de recirculação também possuem uma fila para o acesso concorrente com as demais portas do *switch*. A implementação sequencial realiza a cifração de apenas 512 bits de dados, trabalhando com um campo de *payload* mínimo. A Figura 6 representa o design do algoritmo Forro14 no plano de dados.

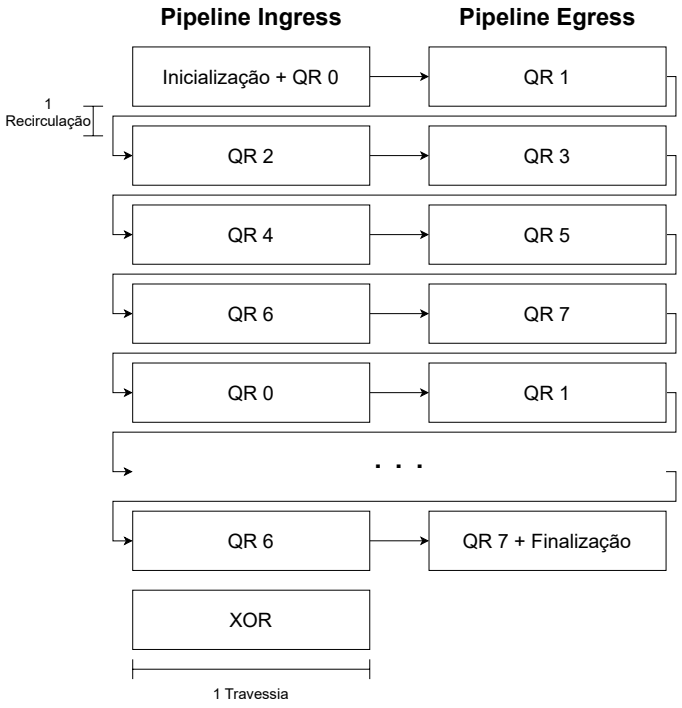

Figura 6. *Design* do Algoritmo Forro14 em Plano de Dados Programável.

Com o objetivo de processar apenas 512 bits, o cabeçalho do pacote para o algoritmo Forro14 foi reduzido em relação ao utilizado pelo ChaCha20 paralelo, conforme Figura 7.

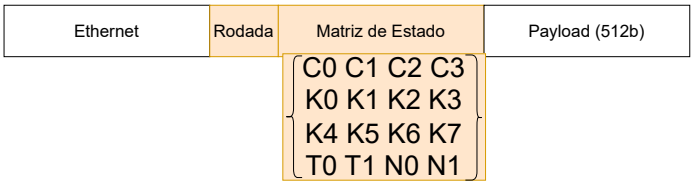

Figura 7. Estrutura de cabeçalho do pacote de processamento do Forro14 em Plano de Dados Programável.

Para uma comparação mais direta com o Forro14 quanto ao uso dos recursos do *switch*, foi feita uma implementação do algoritmo ChaCha20 sequencial, isto é, sem a paralelização dos QRs. Essa implementação foi baseada na implementação do Forro14,

alterando apenas as operações realizadas nos QRs e a quantidade de recirculações feitas para cumprir as 20 rodadas do ChaCha20. Assim como ocorreu no Forro14, o cabeçalho dos pacotes também foi reduzido para o ChaCha20 sequencial. Nesse caso, foram necessárias 40 recirculações com 81 travessias do pipeline para realizar o cálculo da matriz de estados e a cifração.

Para verificar a implementação correta dos algoritmos, foram utilizados os vetores de teste disponíveis na RFC 7539 [Nir and Langley 2015] para o algoritmo ChaCha20 e no repositório de implementação do Forro14 [Coutinho 2023b].

5. Resultados e discussão

5.1. Configuração de testes

Para avaliação do desempenho do algoritmo em plano de dados programável, foi elaborado um cenário com dois *switches* programáveis Tofino: um dos *switches* (TG) está com um gerador de tráfego PIPO-TG [Costa 2023] para gerar tráfegos com cabeçalhos pré-definidos com até 100 Gbps de vazão, enquanto o outro *switch* (SC) está com o algoritmo de cifra de fluxo implementado. A Figura 8 ilustra a configuração de testes.

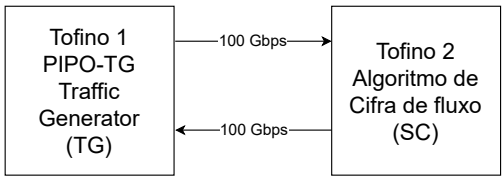

Figura 8. Configuração de testes para avaliação de desempenho.

O tráfego gerado é encaminhado para o *switch* SC (*Stream Cipher)* através de um cabo DAC (*Direct Attached Copper*) de 100 Gbps com conectores QSFP28 e devolvido para o *switch* TG (*Traffic Generator)* através de outro cabo igual. O enlace está configurado para funcionar a 100 Gbps sem FEC (*Forward Error Correction*) e com autonegociação de taxa desabilitada. Para recirculação, o SC possui 2 portas internas que trabalham em taxas de 100 Gbps. Essas duas portas foram utilizadas nos testes para reduzir a perda de pacotes devido à saturação de suas filas, sem ocupar portas externas do *switch*.

Os algoritmos estão com *nonce* e chave definidos de forma estática no código, portanto os valores aleatórios gerados pelo TG como *payload* são decifrados sempre utilizando a mesma chave e *nonce*. Deve ser observado que o *switch* não possui nenhuma estrutura de *cache* para acelerar a decifração com os mesmos parâmetros.

O tamanho do *payload* cifrado nos testes foi sempre de 512 bits. Portanto, todos os algoritmos trabalharam com a mesma quantidade de dados cifrados. Como é feita a decifração de apenas 512 bits, uma única matriz de estado é calculada para cada pacote, sendo os valores dos contadores da matriz sempre mantidos em 0.

5.2. Vazão de tráfego

Para coleta da vazão alcançada após a decifração, foram gerados tráfegos de pacotes de 1 a 20 Gbps e de outros valores comuns de velocidades de enlace de fibra óptica,

Tabela 1. Taxas de vazão dos algoritmos de cifra de fluxo

	Taxa de entrada											
Algoritmo	**1Gbps**	**2Gbps**	**3Gbps**	**4Gbps**	**5Gbps**	**6Gbps**	**7Gbps**	**8Gbps**	**9Gbps**	**10Gbps**	**11Gbps**	**12Gbps**
Forro14	1019	2038	3055	4076	5093	6110	7127	8152	9161	10186	9678	8701
ChaCha20 (Seq)	1019	2038	3055	4076	5093	6110	7127	6418	5581	4932	4310	3893
ChaCha20 (Par)	961	1922	2881	3844	4803	5763	6722	7689	8640	9607	10559	11523
	Taxa de entrada											
Algoritmo	**13Gbps**	**14Gbps**	**15Gbps**	**16Gbps**	**17Gbps**	**18Gbps**	**19Gbps**	**20Gbps**	**25Gbps**	**40Gbps**	**50Gbps**	**100Gbps**
Forro14	7904	7184	6520	5994	6083	5349	5388	4842	3594	1313	912	46
ChaCha20 (Seq)	3606	3145	2747	2233	2787	2336	2009	1860	1163	268	155	2,6
ChaCha20 (Par)	12477	13442	14410	13511	12901	12109	11565	11013	6971	2862	2789	476

como 25 Gbps, 40 Gbps, 50 Gbps e 100 Gbps. O tráfego é medido como o valor inteiro máximo da média móvel de vazão dos últimos 10 segundos na porta de retorno do SC para o TG, produzindo os resultados mostrados na Tabela 1 e na Figura 9.

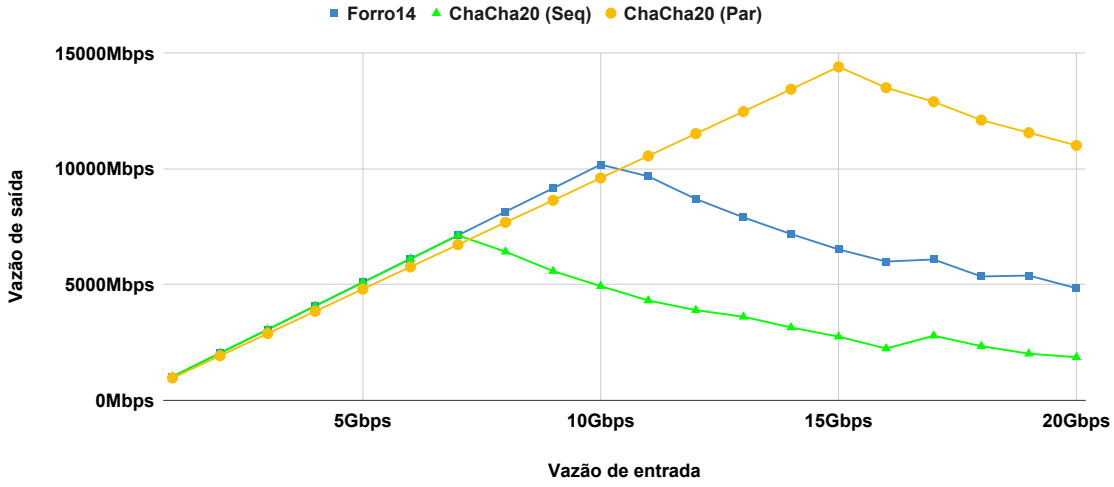

Figura 9. Vazão de saída dos algoritmos de cifra de fluxo no *switch* Tofino para diversas vazões de entrada.

Como demonstrado no gráfico, o algoritmo ChaCha20 paralelo manteve a taxa de transmissão até 15Gbps sem perdas significativas de vazão, enquanto o algoritmo Forro14 chegou até 10 Gbps e o ChaCha20 implementado de forma sequencial até 7 Gbps. Isso se deve principalmente à quantidade de recirculações necessárias para cada algoritmo, visto que mais recirculações implicam em mais pacotes sendo enfileirados nas filas de entrada das portas internas de recirculação.

Nesse ponto, é possível afirmar que o algoritmo ChaCha20 paralelo suporta taxas de tráfego maiores sem perda de pacotes para aplicações focadas no uso de todo o *switch* apenas para cifração/decifração do tráfego. Já o algoritmo Forro14 pode ser utilizado para aplicações que precisem de tráfego de até 10 Gbps sem perda de pacotes com o uso do *switch* apenas para cifração/decifração.

5.3. Ocupação de recursos

Outro ponto a ser verificado é a ocupação de recursos computacionais que cada algoritmo tem no SC. Para a maioria dos cenários de uso, não é interessante que os re-

Tabela 2. Ocupação de recursos dos algoritmos de cifra de fluxo

Componente	ChaCha20 (Par)	Forro14	ChaCha20 (Seq)
ADBB	9,4%	3,6%	2,6%
EMRB	9,9%	12,5%	12,5%
EMSB	9,4%	12,5%	12,5%
GW	9,4%	0,0%	0,0%
HASHB	4,0%	5,4%	5,4%
HASHD	16,7%	2,8%	2,8%
LT-ID	12,5%	12,5%	12,5%
SRAM	1,4%	2,9%	2,7%
STASH	0,5%	12,5%	12,5%
TCAM	1,7%	0,0%	0,0%
TRB	9,4%	0,0%	0,0%
VLIW	4,7%	10,2%	10,2%
EMIX	2,8%	1,8%	1,8%
TMIX	2,1%	0,0%	0,0%
PHV	34,9%	43,1%	43,1%

cursos do *switch* sejam destinados exclusivamente para a cifração de pacotes. Portanto, é relevante avaliar a ocupação de recursos pelos algoritmos.

Para coleta da ocupação de recursos, a Intel disponibiliza a ferramenta *P4 Insight*, que provê uma estimativa do uso dos recursos do *switch* de acordo com o resultado da compilação do código P4 para a ASIC. Diversos recursos compõem a arquitetura da ASIC e sua ocupação pode ser vista na Tabela 2 e no gráfico da Figura 10.

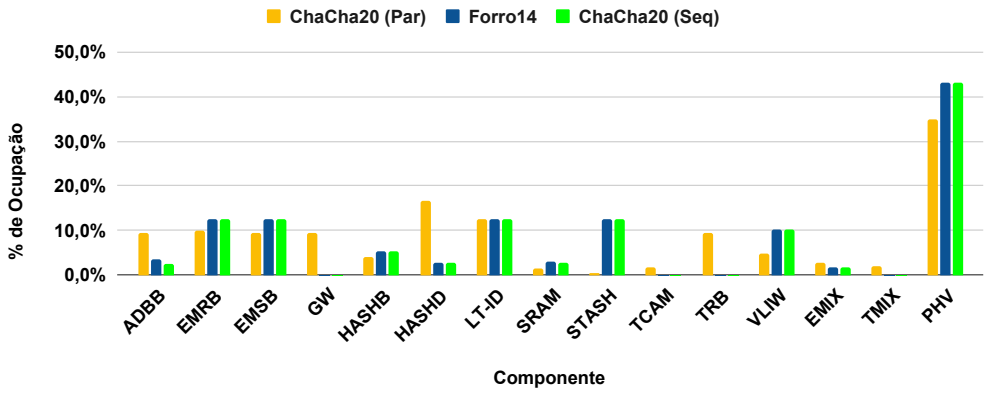

Figura 10. Ocupação de recursos do Tofino pelos algoritmos de cifra de fluxo.

Como destaque da comparação, deve ser citado o *Action Data Bus Bytes* (ADBB) que representa a quantidade de operações que podem ser executadas nos estágios de um pipeline. Todos os algoritmos utilizaram pouco esse recurso, o que é positivo, pois, em *switches* que precisam lidar com muitos protocolos diferentes ou operações especializadas, esse recurso pode se esgotar rapidamente com mais operações concorrentes no *pipeline*.

Ternary Content Addressable Memory (TCAM), *Ternary Result Bus* (TRB) e *Ternary Match Input Crossbar* (TMIX) são recursos utilizados para *matches* ternários e de *longest prefix match* (LPM) em tabelas do *switch*, enquanto *Static RAM* (SRAM), *Exact*

Match Result Bus (EMRB), *Exact Match Search Bus* (EMSB) e *Exact Match Input Crossbar* (EMIX) são utilizados para *matches* exatos em tabelas. Como a memória TCAM é escassa e a SRAM é abundante em switches, as implementações do Forro14 e do ChaCha20 sequencial utilizaram apenas matches exatos. Portanto, não usam a memória TCAM, liberando este importante e escasso recurso para aplicações que necessitam de *match* ternário, como encaminhamento baseado em sub-rede, *multicast* e filtragem de pacotes.

Um ponto curioso é a ocupação dos *Packet Header Vectors* (PHV) do *switch* em cada caso. PHVs são memórias similares a registradores de CPU alocadas em cada estágio de processamento do pipeline para realizar operações aritméticas nos valores coletados dos cabeçalhos. Como o ChaCha20 paralelo realiza mais operações por estágio, espera-se que a alocação de PHVs seja maior que em algoritmos sequenciais que realizam apenas uma operação por estágio. O comportamento observado foi o oposto e uma possível explicação para essa alocação contraintuitiva de PHVs pode estar em eventuais otimizações feitas no algoritmo ChaCha20 paralelo, mas não nos algoritmos sequenciais, tais como: (1) o uso de mais recursos de instrução de *Hash* (*Hash Bit* e *Hash Distribution*, devido ao maior uso da instrução *Hash Identity*) e (2) a forma de alocação dos dados da matriz de estado e do *payload* no momento da cifração/decifração pelo compilador, não utilizando o mesmo registrador para ambas as operações. Logo, é possível que consigamos, em implementações futuras, obter as mesmas alocações otimizadas que foram obtidas pela implementação do ChaCha20 paralelo, o que reforçaria a vantagem do Forro14 em relação ao ChaCha20 no tocante à alocação de escassos recursos do *switch*.

Sendo assim, vemos que a pergunta de pesquisa pode ser respondida da seguinte maneira: em comparação ao ChaCha20 paralelo, o algoritmo Forro14 é a melhor opção para aplicações que demandem até 10 Gbps sem perda de pacotes, visto que possui uma menor utilização de recursos do *switch*, liberando mais recursos para outras funcionalidades. Em cenários onde a ocupação de recursos do *switch* não é relevante, o algoritmo ChaCha20 paralelo apresenta um desempenho melhor em comparação ao Forro14. O ChaCha20 sequencial tem desempenho inferior ao Forro14 para taxas de entrada acima de 7 Gbps. Com relação ao consumo de recursos, vemos (pela Tab. 2) que há um empate entre os dois, com ligeira vantagem para o ChaCha20 sequencial nos recursos ADBB e SRAM. Logo, torna-se mais interessante usar o Forro14 para aplicações cujas taxas de entrada estiverem entre 8 e 10 Gbps.

5.4. Paralelização do algoritmo Forro14

Na proposta do algoritmo Forro14 existe uma sugestão de uma forma de paralelização do algoritmo por meio do cálculo de mais de uma matriz de estado para a cifração de mais de um conjunto de 512 bits de dados por vez [Coutinho 2023a]. Devido a algumas limitações da arquitetura do Tofino quanto à quantidade de dados que podem ser extraídos no *parser* do *pipeline Egress*, essa paralelização não pôde ser implementada ainda, necessitando investigações mais aprofundadas para tentar viabilizá-la. Uma abordagem possível é buscar uma alocação dos valores de estado de forma que sempre se calcule os mesmos elementos de matrizes diferentes em paralelo. A Figura 11 exemplifica uma possível abordagem para essa implementação.

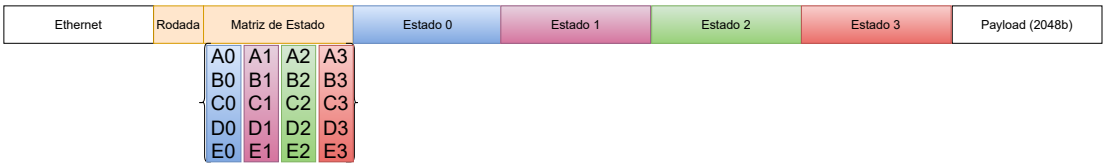

Figura 11. Esquema de cabeçalhos para cálculo de quatro estados do Forro14.

5.5. Impactos da quantidade de portas de recirculação

Em outra frente, investigou-se as taxas que poderiam ser alcançadas se mais portas de recirculações fossem utilizadas. O gráfico da Figura 12 mostra uma relação de vazão alcançada utilizando-se 1, 2 e 4 portas de recirculação para o algoritmo Forro14. Como é possível perceber, a alocação de mais portas de recirculação não tem uma relação linear com a vazão máxima obtida após a cifração.

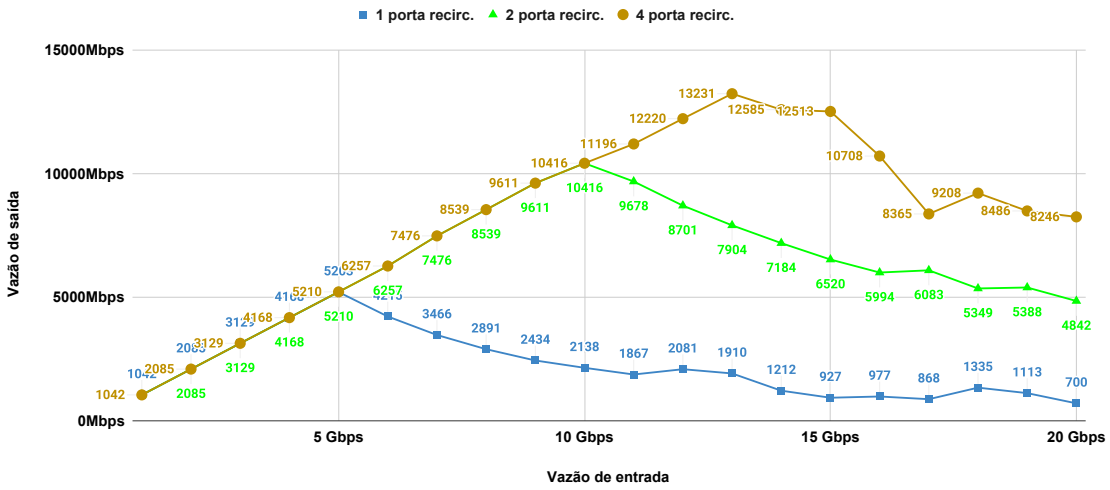

Figura 12. Vazão de saída do Forro14 em função da vazão de entrada e do número de portas de recirculação utilizadas.

Com isso, em uma aplicação na qual o *switch* esteja focado apenas na cifração de tráfego, o uso de até 32 portas de recirculação (2 internas mais 30 externas, para o *switch* utilizado) pode prover aumento expressivo na vazão.

Considerando uma vazão de entrada de 10 Gbps e que o *switch* Tofino trabalha a 100 Gbps, um pacote pode atravessar 10 vezes os *pipelines* antes da chegada do próximo. A partir do momento que o segundo pacote começou a ser processado, ele concorrerá com a travessia do primeiro, fazendo com que ambos atravessem os *pipelines* apenas 5 vezes até a chegada do próximo pacote. Entre as chegadas do terceiro e quarto pacotes, o primeiro pacote atravessará o pipeline 4 vezes, o segundo 3 vezes e o terceiro outras 3 vezes. À medida que mais pacotes vão chegando, essa competição pelos recursos aumenta, reduzindo o número de travessias de cada um durante os intervalos entre as chegadas de novos pacotes. Isso satura as filas de entrada das portas de recirculação, causando a queda da vazão de saída com taxas de entrada maiores.

6. Trabalhos futuros

Para entender melhor as relações entre os fluxos de entrada e saída discutidos acima, seria interessante buscar um modelo matemático que consiga caracterizar a relação entre tais fluxos e o número de portas de recirculação. Tal modelo deveria levar em consideração, entre outros pontos, que o número de recirculações de cada pacote é bem definido para cada algoritmo.

Alguns outros possíveis trabalhos futuros podem ser destacados como, por exemplo, a investigação de uma implementação paralela do Forro14 a partir do cálculo de mais de uma matriz de estado por *pipeline*, investigando-se os impactos em alocação de recursos e vazão. Outro trabalho seria a avaliação do uso de mais portas de recirculação de pacotes para maximizar a operação de cifração e decifração do tráfego pelo *switch*.

Uma outra investigação seria a implementação dos AEADs (*Authenticated Encryption with Associated Data*) XChaCha20-Poly1305 [Arciszewski 2020] e XForro14-Poly1305 [Coutinho et al. 2023a] para verificar os impactos na vazão e na ocupação do *switch* causados pela inclusão de uma garantia de integridade e autenticidade do conteúdo cifrado, a qual é de grande interesse em alguns cenários de ataque.

Considerando o problema em aberto da alocação de PHVs nos algoritmos sequenciais, há ainda o trabalho de reestruturação do código de forma que uma mesma alocação de PHV seja utilizada para o cálculo da matriz de estado e para a operação de cifração e decifração.

7. Conclusão

Este trabalho implementou e avaliou o desempenho do algoritmo de cifra de fluxo Forro14 em plano de dados programável da arquitetura Intel Tofino.

A implementação foi comparada com duas versões do algoritmo ChaCha20 (paralela e sequencial).

Apesar de não conseguir alcançar as mesmas vazões de dados que a versão paralela do ChaCha20, o Forro14 usa menos recursos do *switch* e é a melhor opção para taxas de até 10Gbps. Além disso, ele supera a versão sequencial do ChaCha20 em termos de capacidade de vazão, utilizando aproximadamente os mesmos recursos que este.

Como o Forro14 é mais econômico em termos de demandas por recursos e oferece uma taxa de vazão satisfatória para vários casos de uso, ele se mostra uma opção interessante em aplicações que necessitam da implementação de outras funções no mesmo *switch*, como roteamento e filtragem de pacotes, por exemplo.

8. Agradecimentos

Agradecemos ao Fabricio Rodríguez e ao Francisco Vogt do Information and Networking Technologies Research and Innovation Group (INTRIG) do DCA da UNICAMP pelas contribuições e discussões relevantes para a implementação dos algoritmos Forro14 e ChaCha20 sequencial no hardware Intel Tofino.

Este trabalho foi parcialmente financiado pela Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP), processo 2021/00199-8, CPE SMARTNESS.

Referências

Arciszewski, S. (2020). XChaCha: eXtended-nonce ChaCha and AEAD_XChaCha20_Poly1305. Internet-Draft draft-irtf-cfrg-xchacha-03, Internet Engineering Task Force. Work in Progress.

Bernstein, D. J. et al. (2008). Chacha, a variant of salsa20. In *Workshop record of SASC*, volume 8, pages 3–5. Citeseer.

Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., et al. (2014). P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95.

Chen, X. (2020). Implementing aes encryption on programmable switches via scrambled lookup tables. In *Proceedings of the Workshop on Secure Programmable Network Infrastructure*, pages 8–14.

Costa, F. G. (2023). Pipo-tg: parameterizable high performance traffic generation.

Coutinho, M. (2023a). Design, diffusion, and cryptanalysis of symmetric primitive.

Coutinho, M. (2023b). forro_cipher. `https://github.com/murcoutinho/forro_cipher`. Online: Acesso em 28-05-2024.

Coutinho, M., Passos, I., and Borges, F. (2023a). The design and implementation of xforró14-poly1305: a new authenticated encryption scheme. In *Anais do XXIII Simpósio Brasileiro de Segurança da Informação e de Sistemas Computacionais*, pages 456–469, Porto Alegre, RS, Brasil. SBC.

Coutinho, M., Passos, I., Vásquez, J. C. G., Sarkar, S., de Mendonça, F. L., de Sousa Jr, R. T., and Borges, F. (2023b). Latin dances reloaded: Improved cryptanalysis against salsa and chacha, and the proposal of forró. *Journal of Cryptology*, 36(3):18.

Dang, H. T., Bressana, P., Wang, H., Lee, K. S., Zilberman, N., Weatherspoon, H., Canini, M., Pedone, F., and Soulé, R. (2020). P4xos: Consensus as a network service. *IEEE/ACM Transactions on Networking*, 28(4):1726–1738.

Datta, R., Choi, S., Chowdhary, A., and Park, Y. (2018). P4guard: Designing p4 based firewall. In *MILCOM 2018-2018 IEEE Military Communications Conference (MILCOM)*, pages 1–6. IEEE.

Dworkin, M., Barker, E., Nechvatal, J., Foti, J., Bassham, L., Roback, E., and Dray, J. (2001). Advanced encryption standard (aes).

Fernandes, E. L. and Rothenberg, C. E. (2014). Openflow 1.3 software switch. *Salao de Ferramentas do XXXII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuıdos SBRC*, pages 1021–1028.

Hauser, F., Häberle, M., Merling, D., Lindner, S., Gurevich, V., Zeiger, F., Frank, R., and Menth, M. (2023). A survey on data plane programming with p4: Fundamentals, advances, and applied research. *Journal of Network and Computer Applications*, 212:103561.

Jin, X., Li, X., Zhang, H., Soulé, R., Lee, J., Foster, N., Kim, C., and Stoica, I. (2017). Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 121–136.

Kfoury, E. F., Crichigno, J., and Bou-Harb, E. (2021). An exhaustive survey on p4 programmable data plane switches: Taxonomy, applications, challenges, and future trends. *IEEE Access*, 9:87094–87155.

Kreutz, D., Ramos, F. M., Verissimo, P. E., Rothenberg, C. E., Azodolmolky, S., and Uhlig, S. (2014). Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1):14–76.

Li, G., Zhang, M., Liu, C., Kong, X., Chen, A., Gu, G., and Duan, H. (2019). Nethcf: Enabling line-rate and adaptive spoofed ip traffic filtering. In *2019 IEEE 27th international conference on network protocols (ICNP)*, pages 1–12. IEEE.

Mahrach, S., Mjihil, O., and Haqiq, A. (2018). Scalable and dynamic network intrusion detection and prevention system. In *Innovations in Bio-Inspired Computing and Applications: Proceedings of the 8th International Conference on Innovations in Bio-Inspired Computing and Applications (IBICA 2017) held in Marrakech, Morocco, December 11-13, 2017*, pages 318–328. Springer.

Nir, Y. and Langley, A. (2015). ChaCha20 and Poly1305 for IETF Protocols. RFC 7539.

Peterson, L., Cascone, C., and Davie, B. (2021). *Software-Defined Networks: A Systems Approach*. Systems Approach LLC.

Scholz, D., Oeldemann, A., Geyer, F., Gallenmüller, S., Stubbe, H., Wild, T., Herkersdorf, A., and Carle, G. (2019). Cryptographic hashing in p4 data planes. In *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 1–6. IEEE.

Sivaraman, V., Narayana, S., Rottenstreich, O., Muthukrishnan, S., and Rexford, J. (2017). Heavy-hitter detection entirely in the data plane. In *Proceedings of the Symposium on SDN Research*, pages 164–176.

Tokusashi, Y., Matsutani, H., and Zilberman, N. (2018). Lake: the power of in-network computing. In *2018 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–8. IEEE.

Vieira, M. A., Castanho, M. S., Pacífico, R. D., Santos, E. R., Júnior, E. P. C., and Vieira, L. F. (2020). Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications. *ACM Computing Surveys (CSUR)*, 53(1):1–36.

Yoo, S. and Chen, X. (2021). Secure keyed hashing on programmable switches. In *Proceedings of the ACM SIGCOMM 2021 Workshop on Secure Programmable network INfrastructure*, pages 16–22.

Yoshinaka, Y., Takemasa, J., Koizumi, Y., and Hasegawa, T. (2022). On implementing chacha on a programmable switch. In *Proceedings of the 5th International Workshop on P4 in Europe*, pages 15–18.

Zheng, C., Rienecker, B., and Zilberman, N. (2023). Qcmp: Load balancing via in-network reinforcement learning. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Future of Internet Routing & Addressing*, pages 35–40.